CHAPTER 6

Optimization

This chapter builds on Chapter 5 about equation solving and explores the related topic of solving optimization problems. Optimization is the process of finding and selecting the optimal element from a set of feasible candidates. In mathematical optimization, this problem is usually formulated as determining the extreme value of a function on a given domain. An extreme value, or an optimal value, can refer to either the minimum or maximum of the function, depending on the application and the specific problem. The chapter covers the optimization of real-valued functions of one or several variables, which optionally can be subject to constraints that restrict the function's domain.

The applications of mathematical optimization are many and varied, and so are the methods and algorithms that must be employed to solve optimization problems. Since optimization is a universally important mathematical tool, it has been developed and adapted for use in many fields of science and engineering, and the terminology used to describe optimization problems varies between fields. For example, the optimized mathematical function may be called a cost function, loss function, energy function, or objective function, to mention a few. Here, we use the generic term objective function.

Optimization is closely related to equation solving because at an optimal value of a function, its derivative, or gradient in the multivariate case, is zero. The converse, however, is not necessarily true. But one method to solve optimization problems is to solve for the zeros of the derivative or the gradient and test the resulting candidates for optimality. Although this approach is only sometimes feasible and often requires other numerical techniques, many of these are closely related to the numerical methods for root finding covered in Chapter 5.

This chapter discusses using SciPy's `optimize` optimization module for nonlinear optimization problems, and it briefly explores using `cvxopt`, the convex optimization library for linear optimization problems with linear constraints. This library also has powerful solvers for quadratic programming problems.

---

cvxopt This convex optimization library provides solvers for linear and quadratic optimization problems. At the time of writing, the latest version is 1.3.2. For more information, visit the project's website at `http://cvxopt.org`. This chapter used the library for constrained linear optimization.

Importing Modules

Let's use the `optimize` module from the SciPy library again, assuming that this module is imported in the following manner.

```
In [1]: from scipy import optimize
```

R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7_6

The latter part of this chapter looks at linear programming using the `cvxopt` library, which is assumed to be entirely imported without any alias.

```
In [2]: import cvxopt
```

For basic numerics, symbolics, and plotting, let's use the NumPy, SymPy, and Matplotlib libraries, which are imported and initialized using the conventions introduced in earlier chapters.

```
In [3]: import matplotlib.pyplot as plt
In [4]: import numpy as np
In [5]: import sympy
In [6]: sympy.init_printing()
```

Classification of Optimization Problems

Let's restrict our attention to mathematical optimization of real-valued functions with one or more dependent variables. While many mathematical optimization problems can be formulated in this way, a notable exception is the optimization of functions over discrete variables, for example, integers, which is beyond the scope of this book.

A general optimization problem of the type considered here can be formulated as a minimization problem, $\min_x f(x)$, subject to sets of m equality constraints $g(x) = 0$ and p inequality constraints $h(x) \leq 0$. Here $f(x)$ is a real-valued function of x, which can be a scalar or a vector $x = (x_0, x_1, \ldots, x_n)^T$, while $g(x)$ and $h(x)$ can be vector-valued functions: $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, $g : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and $h : \mathbb{R}^n \longrightarrow \mathbb{R}^p$. Note that maximizing $f(x)$ is equivalent to minimizing $-f(x)$, so without loss of generality, it is sufficient to consider only minimization problems.

Depending on the properties of the objective function $f(x)$ and the equality and inequality constraints $g(x)$ and $h(x)$, this formulation includes a wide variety of problems. A general mathematical optimization on this form is difficult to solve, and there are no efficient methods for solving completely generic optimization problems. However, there are efficient methods for many important special cases. In optimization, it is therefore important to know as much as possible about the objective functions and constraints to solve a problem.

Optimization problems are classified depending on the properties of the $f(x)$, $g(x)$, and $h(x)$ functions. First and foremost, the problem is *univariate* or *one-dimensional* if x is a scalar, $x \in \mathbb{R}$, and *multivariate* or *multidimensional* if x is a vector, $x \in \mathbb{R}^n$. For high-dimensional objective functions with larger n, the optimization problem is harder and more computationally demanding to solve. If the objective function and the constraints are all linear, the problem is a linear optimization problem or a *linear programming* problem.[1] If either the objective function or the constraints are nonlinear, it is a nonlinear optimization problem or a *nonlinear programming* problem. With respect to constraints, important subclasses of optimization are unconstrained problems and those with linear and nonlinear constraints. Finally, handling equality and inequality constraints requires different approaches.

As usual, nonlinear problems are much harder to solve than linear problems because they have a wider variety of possible behaviors. A general nonlinear problem can have both local and global minima, making it very difficult to find the global minima. Iterative solvers may often converge to local minima rather than the global minima or may even fail to converge altogether if there are both local and global minima. However, an important subclass of nonlinear problems that can be solved efficiently is *convex problems*. Convexity is directly related to the absence of strictly local minima and the existence of a unique global minimum.

[1] For historical reasons, optimization problems are often referred to as *programming* problems, which are not directly related to computer programming in the modern sense.

By definition, a function is convex on an interval $[a, b]$ if the values of the function on this interval lie below the line through the endpoints $(a, f(a))$ and $(b, f(b))$. This condition, which can be readily generalized to the multivariate case, implies several important properties, such as a unique minimum on the interval. Because of strong properties like this one, convex problems can be solved efficiently despite being nonlinear. The concepts of local and global minima and convex and nonconvex functions are illustrated in Figure 6-1.

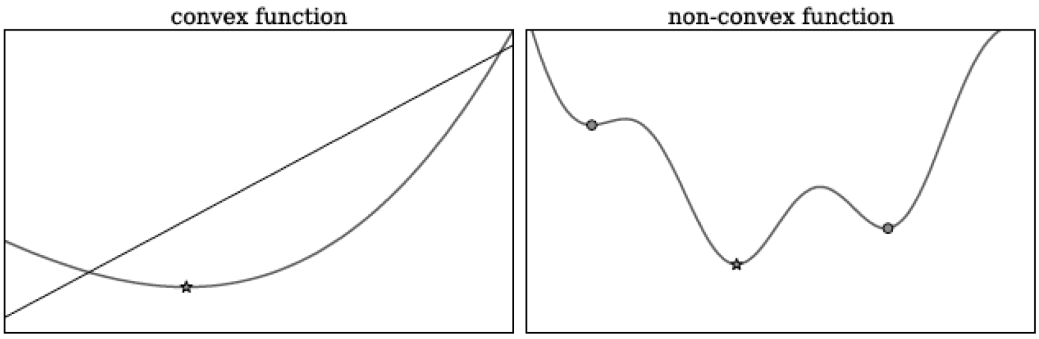

***Figure 6-1.** Illustration of a convex function (left) and a nonconvex function (right) with a global minimum and two local minima*

Whether the objective function $f(x)$ and the constraints $g(x)$ and $h(x)$ are continuous and smooth are properties that have significant implications for the methods and techniques that can be used to solve an optimization problem. Discontinuities in these functions, or their derivatives or gradients, cause difficulties for many of the available optimization methods. The following assumes these functions are continuous and smooth. On a related note, if the function itself is not known exactly but contains noise due to measurements or other reasons, many of the methods discussed in the following may not be suitable.

Optimization of continuous and smooth functions is closely related to nonlinear equation solving because extremal values of an $f(x)$ function correspond to points where its derivative, or gradient, is zero. Finding candidates for the optimal value of $f(x)$ is therefore equivalent to solving the (in general nonlinear) equation system $\nabla f(x) = 0$. However, a solution to $\nabla f(x) = 0$, known as a stationary point, does not necessarily correspond to a minimum of $f(x)$; it can also be a maximum or a saddle point; see Figure 6-2. Therefore, candidates obtained by solving $\nabla f(x) = 0$ should be tested for optimality. For unconstrained objective functions, the higher-order derivatives, or the Hessian matrix,

$$\left\{H_f(x)\right\}_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j},$$

for the multivariate case, can be used to determine if a stationary point is a local minimum. If the second-order derivative is positive, or the Hessian positive definite, when evaluated at stationary point x^*, then x^* is a local minimum. A negative second-order derivative, or negative definite Hessian, corresponds to a local maximum, and a zero second-order derivative, or an indefinite Hessian, corresponds to saddle point.

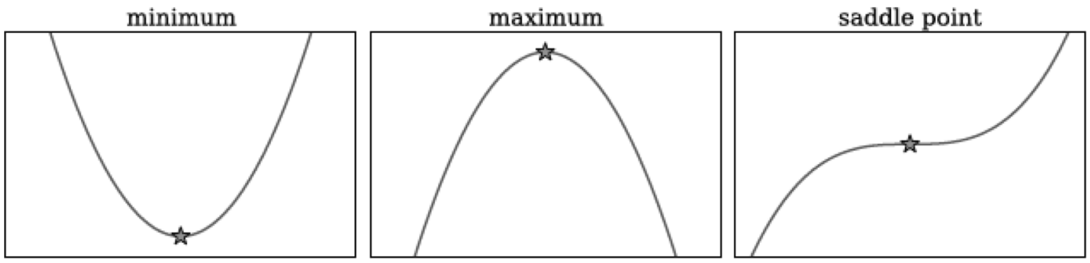

Figure 6-2. *Illustration of different stationary points of a one-dimensional function*

Algebraically solving the equation system $\nabla f(x) = 0$ and testing the candidate solutions for optimality is one possible strategy for solving an optimization problem. However, it is not always a feasible method. We may not have an analytical expression for $f(x)$ from which we can compute the derivatives, and the resulting nonlinear equation system may not be easy to solve, especially not to find all its roots. There are alternative numerical optimization approaches for such cases, some of which have analogs among the root-finding methods discussed in Chapter 5. The remaining part of this chapter explores the various classes of optimization problems and how such problems can be solved using available optimization libraries for Python.

Univariate Optimization

Optimization of a function that only depends on a single variable is relatively easy. In addition to the analytical approach of seeking the roots of the function's derivative, we can employ techniques similar to the root-finding methods for univariate functions, namely, bracketing methods and Newton's method. Like the bisection method for univariate root finding, it is possible to use bracketing and iteratively refine an interval using function evaluations alone. Refining an interval $[a, b]$ that contains a minimum can be achieved by evaluating the function at two interior points x_1 and x_2, $x_1 < x_2$, and selecting $[x_1, b]$ as a new interval if $f(x_1) > f(x_2)$, and $[a, x_2]$ otherwise. This idea is used in the *golden section search* method, which additionally uses the trick of choosing x_1 and x_2 such that their relative positions in the $[a, b]$ interval satisfy the golden ratio. This has the advantage of allowing the reuse of one function evaluation from the previous iteration and thus only requiring one new function evaluation in each iteration and reducing the interval with a constant factor. This approach is guaranteed to converge to an optimal point for functions with a unique minimum on the given interval, but this is not guaranteed for more complicated functions. It is, therefore, important to carefully select the initial interval, ideally relatively close to an optimal point. In the SciPy `optimize` module, the `golden` function implements the golden search method.

Like the bisection method for root finding, the golden search method is a (relatively) reliable but slowly converging method. Methods with better convergence can be constructed if the values of the function evaluations are used rather than only comparing the values to each other (similar to using only the sign of the functions, as in the bisection method). The function values can be used to fit a polynomial, for example, a quadratic polynomial, which can be interpolated to find a new approximation for the minimum, resulting in a candidate for a new function evaluation, after which the process can be iterated. This approach can converge faster but is riskier than bracketing and may not converge at all or converge to local minima outside the given bracket interval.

Newton's method for root finding is an example of a quadratic approximation method that can be applied to find a function minimum by applying the method to the derivative rather than the function itself. This yields the iteration formula $x_{k+1} = x_k - f'(x_k)/f'(x_k)$, which can quickly converge if started close to an optimal point but may not converge at all if started too far from the optimal value. This formula also requires evaluating both the derivative and the second-order derivative in each iteration. This can be a suitable method if analytical expressions for these derivatives are available. If only function evaluations are available, the derivatives may be approximated using an analog of the secant method for root finding.

A combination of the two previous methods is typically used in practical implementations of univariate optimization routines, giving both stability and fast convergence. In SciPy's `optimize` module, the `brent` function is such a hybrid method, and it is generally the preferred method for optimizing univariate functions with SciPy. This variant of the golden section search method uses inverse parabolic interpolation to obtain faster convergence.

Instead of calling the `optimize.golden` and `optimize.brent` functions directly; it is convenient to use the unified interface function `optimize.minimize_scalar`, which dispatches to the `optimize.golden` and `optimize.brent` functions depending on the value of the `method` keyword argument, where the currently allowed options are `'Golden'`, `'Brent'`, or `'Bounded'`. The last option dispatches to `optimize.fminbound`, which performs optimization on a bounded interval, corresponding to an optimization problem with inequality constraints that limit the domain of objective function $f(x)$. Note that the `optimize.golden` and `optimize.brent` functions may converge to a local minimum outside the initial bracket interval, but `optimize.fminbound` would return the value at the end of the allowed range in such cases.

As an example for illustrating these techniques, consider the following classic optimization problem: Minimize the area of a cylinder with unit volume. Here, suitable variables are the radius r and height h of the cylinder, and the objective function is $f([r, h]) = 2\pi r^2+2\pi rh$, subject to the equality constraint $g([r, h]) = \pi r^2 h - 1 = 0$. This problem is formulated here as a two-dimensional optimization problem with an equality constraint. However, we can algebraically solve the constraint equation for one of the dependent variables, for example, $h = 1/\pi r^2$, and substitute this into the objective function to obtain an unconstrained one-dimensional optimization problem: $f(r) = 2\pi r^2+2/r$. To begin with, we can solve this problem symbolically using SymPy, using the method of equating the derivative of $f(r)$ to zero.

```
In [7]: r, h = sympy.symbols("r, h")
In [8]: Area = 2 * sympy.pi * r**2 + 2 * sympy.pi * r * h
In [9]: Volume = sympy.pi * r**2 * h
In [10]: h_r = sympy.solve(Volume - 1)[0]
In [11]: Area_r = Area.subs(h_r)
In [12]: rsol = sympy.solve(Area_r.diff(r))[0]
In [13]: rsol
```

Out[13]: $\frac{2^{2/3}}{2\sqrt[3]{\pi}}$

```
In [14]: _.evalf()
Out[14]: 0.541926070139289
```

Now verify that the second derivative is positive and that `rsol` corresponds to a minimum.

```
In [15]: Area_r.diff(r, 2).subs(r, rsol)
```

Out[15]: 12π

```
In [16]: Area_r.subs(r, rsol)
```

Out[16]: $3\sqrt[3]{2\pi}$

```
In [17]: _.evalf()
Out[17]: 5.53581044593209
```

This approach is often feasible for simple problems, but for more realistic problems, we typically need to resort to numerical techniques. To solve this problem using SciPy's numerical optimization functions, we first define a Python function `f` that implements the objective function. To solve the optimization problem, we then pass this function to, for example, `optimize.brent`. Optionally, we can use the `brack` keyword argument to specify a starting interval for the algorithm.

```
In [18]: def f(r):
    ...:     return 2 * np.pi * r**2 + 2 / r
In [19]: r_min = optimize.brent(f, brack=(0.1, 4))
In [20]: r_min
Out[20]: 0.541926077256
In [21]: f(r_min)
Out[21]: 5.53581044593
```

Instead of calling `optimize.brent` directly, we could use the generic interface for scalar minimization problems `optimize.minimize_scalar`. Note that we must use the bracket keyword argument to specify a starting interval in this case.

```
In [22]: optimize.minimize_scalar(f, bracket=(0.1, 4))
Out[22]:  nit: 13
          fun: 5.5358104459320856
            x: 0.54192606489766715
         nfev: 14
```

All these methods give that the radius that minimizes the area of the cylinder is approximately 0.54 (the exact result from the symbolic calculation is $2^{2/3}/2\sqrt[3]{\pi}$) and a minimum area of approximately 5.54 (the exact result is $3\sqrt[3]{2\pi}$). The objective function we minimized in this example is plotted in Figure 6-3, where the minimum is marked with a red star. It is a good idea to visualize the objective function before attempting a numerical optimization because it can help identify a suitable initial interval or a starting point for the numerical optimization routine.

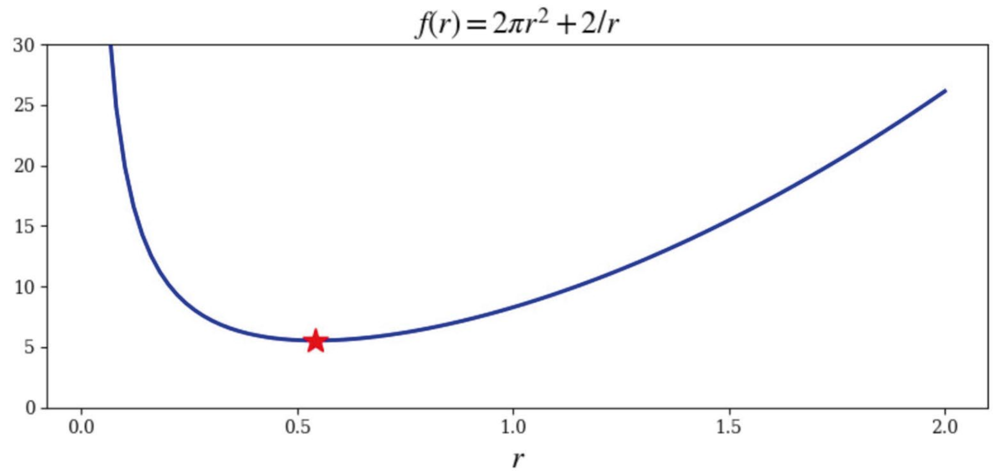

Figure 6-3. *The surface area of a cylinder with unit volume as a function of the radius r. A star denotes the location of the minimum*

Unconstrained Multivariate Optimization

Multivariate optimization is significantly more complex than the univariate optimization discussed in the previous section. In particular, the analytical approach of solving the nonlinear equations for roots of the gradient is rarely feasible in the multivariate case, and the bracketing scheme used in the golden search method is also not directly applicable. Instead, we must resort to techniques that start at some point in the coordinate space and use different strategies to move toward a better approximation of the minimum point. The most basic approach of this type is to consider the gradient $\nabla f(x)$ of the objective function $f(x)$ at a given point x. In general, the negative of the gradient, $-\nabla f(x)$, always points in the direction in which the $f(x)$ function decreases the most. As a minimization strategy, it is therefore sensible to move along this direction for some distance α_k and then iterate this scheme at the new point. This method is known as the *steepest descent method*, and it gives the iteration formula $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$, where α_k is a free parameter known as the *line search parameter* that describes how far along the given direction to move in each iteration. An appropriate α_k can, for example, be selected by solving the one-dimensional optimization problem $\min_{\alpha_k} f(x_k - \alpha_k \nabla f(x_k))$. This method is guaranteed to progress and eventually converge to a minimum of the function. But the convergence can be quite slow because this method tends to overshoot along the direction of the gradient, giving a zigzag approach to the minimum. Nonetheless, the steepest descent method is the conceptual basis for many multivariate optimization algorithms, and with suitable modifications, the convergence can be sped up.

Newton's method for multivariate optimization is a modification of the steepest descent method that can improve convergence. As in the univariate case, Newton's method can be viewed as a local quadratic approximation of the function, which, when minimized, gives an iteration scheme. In the multivariate case, the iteration formula is $x_{k+1} = x_k - H_f^{-1}(x_k)\nabla f(x_k)$, where compared to the steepest descent method, the gradient has been replaced with the gradient multiplied from the left with the inverse of the Hessian matrix for the function.[2] In general, this alters both the direction and the length of the step, so this method is not strictly the steepest descent method and may not converge if started too far from a minimum. However, when close to a minimum, it converges quickly. As usual, there is a trade-off between convergence rate and stability. As formulated here, Newton's method requires both the gradient and the Hessian of the function.

In SciPy, Newton's method is implemented in the `optimize.fmin_ncg` function. This function takes the following arguments: a Python function for the objective function, a starting point, a Python function for evaluating the gradient, and (optionally) a Python function for evaluating the Hessian. To see how this method can be used to solve an optimization problem, consider the following problem: $\min_x f(x)$ where the objective function is $f(x) = (x_1 - 1)^4 + 5(x_2 - 1)^2 - 2x_1x_2$. To apply Newton's method, we need to calculate the gradient and the Hessian. For this case, the calculation can easily be done by hand. However, for generality, the following uses SymPy to compute symbolic expressions for the gradient and the Hessian. To this end, let's begin by defining symbols and a symbolic expression for the objective function and then use the `sympy.diff` function for each variable to obtain the gradient and Hessian in symbolic form.

```
In [23]: x1, x2 = sympy.symbols("x_1, x_2")
In [24]: f_sym = (x1-1)**4 + 5 * (x2-1)**2 - 2*x1*x2
In [25]: fprime_sym = [f_sym.diff(x_) for x_ in (x1, x2)]
In [26]: # Gradient
    ...: sympy.Matrix(fprime_sym)
```

Out[26]: $\begin{bmatrix} -2x_2 + 4(x_1 - 1)^3 \\ -2x_1 + 10x_2 - 10 \end{bmatrix}$

```
In [27]: fhess_sym = [[f_sym.diff(x1_, x2_) for x1_ in (x1, x2)]
    ...:              for x2_ in (x1, x2)]
In [28]: # Hessian
    ...: sympy.Matrix(fhess_sym)
```

Out[28]: $\begin{bmatrix} 12(x_1 - 1)^2 & -2 \\ -2 & 10 \end{bmatrix}$

[2] In practice, the inverse of the Hessian does not need to be computed, and instead we can solve the linear equation system $H_f(x_k)y_k = -\nabla f(x_k)$ and use the integration formula $x_{k+1} = x_k + y_k$.

Now that there is a symbolic expression for the gradient and the Hessian, we can create vectorized functions for these expressions using `sympy.lambdify`.

```
In [29]: f_lmbda = sympy.lambdify((x1, x2), f_sym, 'numpy')
In [30]: fprime_lmbda = sympy.lambdify((x1, x2), fprime_sym, 'numpy')
In [31]: fhess_lmbda = sympy.lambdify((x1, x2), fhess_sym, 'numpy')
```

However, the functions produced by `sympy.lambdify` take one argument for each variable in the corresponding expression, and the SciPy optimization functions expect a vectorized function where all coordinates are packed into one array. To obtain functions compatible with the SciPy optimization routines, we wrap each of the functions generated by `sympy.lambdify` with a Python function that rearranges the arguments.

```
In [32]: def func_XY_to_X_Y(f):
    ...:     """
    ...:     Wrapper for f(X) -> f(X[0], X[1])
    ...:     """
    ...:     return lambda X: np.array(f(X[0], X[1]))
In [33]: f = func_XY_to_X_Y(f_lmbda)
In [34]: fprime = func_XY_to_X_Y(fprime_lmbda)
In [35]: fhess = func_XY_to_X_Y(fhess_lmbda)
```

Now the `f`, `fprime`, and `fhess` functions are vectorized Python functions on the form that `optimize.fmin_ncg` expects, and we can proceed with a numerical optimization of the problem at hand by calling this function. In addition to the functions prepared from SymPy expressions, we must also give a starting point for the Newton method. Let's use (0, 0) as the starting point.

```
In [36]: x_opt = optimize.fmin_ncg(f, (0, 0), fprime=fprime, fhess=fhess)
         Optimization terminated successfully.
              Current function value: -3.867223
              Iterations: 8
              Function evaluations: 10
              Gradient evaluations: 17
              Hessian evaluations: 8
In [37]: x_opt
Out[37]: array([ 1.88292613,  1.37658523])
```

The routine found a minimum point at $(x_1, x_2) = (1.88292613, 1.37658523)$, and diagnostic information about the solution was also printed to standard output, including the number of iterations and the number of function, gradient, and Hessian evaluations that were required to arrive at the solution. As usual, it is illustrative to visualize the objective function $g(\beta) = \sum_{i=0}^{m} r_i(\beta)^2 = \|r(\beta)\|^2$ and the solution (see Figure 6-4).

```
In [38]: fig, ax = plt.subplots(figsize=(6, 4))
    ...: x_ = y_ = np.linspace(-1, 4, 100)
    ...: X, Y = np.meshgrid(x_, y_)
    ...: c = ax.contour(X, Y, f_lmbda(X, Y), 50)
    ...: ax.plot(x_opt[0], x_opt[1], 'r*', markersize=15)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: plt.colorbar(c, ax=ax)
```

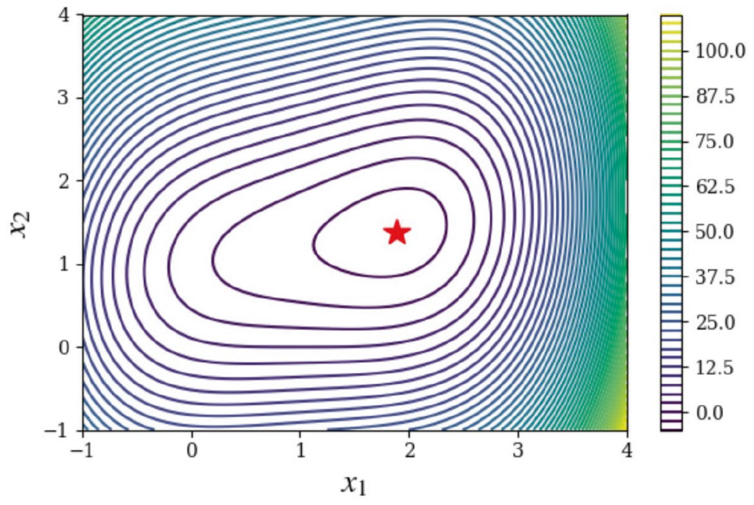

Figure 6-4. *Contour plot of the objective function $f(x) = (x_1 - 1)^4 + 5(x_2 - 1)^2 - 2x_1x_2$. A red star marks the minimum point*

In practice, it may not always be possible to provide functions for evaluating both the gradient and the Hessian of the objective function, and often, it is convenient with a solver that only requires function evaluations. For such cases, several methods exist to estimate the gradient, Hessian, or both numerically. Methods that approximate the Hessian are known as quasi-Newton methods, and there are also alternative iterative methods that avoid entirely using the Hessian. Two popular methods are the Broyden-Fletcher-Goldfarb-Shanno (BFGS) and the conjugate gradient methods, which are implemented in SciPy as the `optimize.fmin_bfgs` and `optimize.fmin_cg` functions. The BFGS method is a quasi-Newton method that can gradually build up numerical estimates of the Hessian and the gradient, if necessary. The conjugate gradient method is a variant of the steepest descent method and does not use the Hessian. It can be used with numerical estimates of the gradient obtained from only function evaluations. With these methods, the number of function evaluations required to solve a problem is much larger than for Newton's method, which, on the other hand, also evaluates the gradient and the Hessian. Both `optimize.fmin_bfgs` and `optimize.fmin_cg` can optionally accept a function for evaluating the gradient, but if not provided, the gradient is estimated from function evaluations.

The preceding problem, which was solved with the Newton method, can also be solved using the `optimize.fmin_bfgs` and `optimize.fmin_cg`, without providing a function for the Hessian.

```
In [39]: x_opt = optimize.fmin_bfgs(f, (0, 0), fprime=fprime)
         Optimization terminated successfully.
             Current function value: -3.867223
             Iterations: 10
             Function evaluations: 14
             Gradient evaluations: 14
In [40]: x_opt
Out[40]: array([ 1.88292605,  1.37658523])
In [41]: x_opt = optimize.fmin_cg(f, (0, 0), fprime=fprime)
         Optimization terminated successfully.
             Current function value: -3.867223
             Iterations: 7
             Function evaluations: 17
             Gradient evaluations: 17
In [42]: x_opt
Out[42]: array([ 1.88292613,  1.37658522])
```

Note that the number of function and gradient evaluations is larger than for Newton's method, as shown in the diagnostic output from the optimization solvers in the previous examples. As mentioned, both methods can also be used without providing a function for the gradient, as shown in the following example using the `optimize.fmin_bfgs` solver.

```
In [43]: x_opt = optimize.fmin_bfgs(f, (0, 0))
         Optimization terminated successfully.
             Current function value: -3.867223
             Iterations: 10
             Function evaluations: 56
             Gradient evaluations: 14
In [44]: x_opt
Out[44]: array([ 1.88292604,  1.37658522])
```

In this case, the number of function evaluations is even larger, but it is convenient not to implement functions for the gradient and the Hessian.

In general, the BFGS method is often a good first approach to try, particularly if neither the gradient nor the Hessian is known. If only the gradient is known, then the BFGS method is still the generally recommended method, although the conjugate gradient method is often a competitive alternative to the BFGS method. If both the gradient and the Hessian are known, Newton's method is the method with the fastest convergence in general. However, it should be noted that although the BFGS and the conjugate gradient methods theoretically have slower convergence than Newton's method, they can sometimes offer improved stability and be preferable. Each iteration can also be more computationally demanding with Newton's method compared to quasi-Newton methods and the conjugate gradient method, and especially for large problems, these methods can be faster despite requiring more iterations.

The methods for multivariate optimization discussed so far converge to a local minimum. Problems with many local minima can easily lead to a situation when the solver easily gets stuck in a local minimum, even if a global minimum exists. Although there is no complete and general solution to this problem, a practical approach to partially alleviate this problem is using a brute-force search over a coordinate grid to find a suitable starting point for an iterative solver. At least this gives a systematic approach to finding a global minimum within given coordinate ranges. In SciPy, the `optimize.brute` function can perform a systematic search. To illustrate this method, consider the problem of minimizing the $4 \sin x\pi + 6 \sin y\pi + (x - 1)^2 + (y - 1)^2$ function, which has a large number of local minima. This can make it tricky to pick a suitable initial point for an iterative solver. To solve this optimization problem with SciPy, define a Python function for the objective function.

```
In [45]: def f(X):
   ...:     x, y = X
   ...:      return (4 * np.sin(np.pi * x) +
   ...:              6 * np.sin(np.pi * y)) + (x - 1)**2 + (y - 1)**2
```

To systematically search for the minimum over a coordinate grid, we call `optimize.brute` with the objective function `f` as the first parameter and a tuple of `slice` objects as the second argument, one for each coordinate. The `slice` objects specify the coordinate grid to search for a minimum value. Let's also set the keyword argument `finish=None`, which prevents the `optimize.brute` from automatically refining the best candidate.

```
In [46]: x_start = optimize.brute(
    ...:     f, (slice(-3, 5, 0.5), slice(-3, 5, 0.5)), finish=None)
In [47]: x_start
Out[47]: array([ 1.5,  1.5])
In [48]: f(x_start)
Out[48]: -9.5
```

On the coordinate grid specified by the given tuple of `slice` objects, the optimal point is $(x_1, x_2) = (1.5, 1.5)$, with a corresponding objective function minimum of –9.5. This is now a good starting point for a more sophisticated iterative solver, such as `optimize.fmin_bfgs`.

```
In [49]: x_opt = optimize.fmin_bfgs(f, x_start)
         Optimization terminated successfully.
              Current function value: -9.520229
              Iterations: 4
              Function evaluations: 28
              Gradient evaluations: 7
In [50]: x_opt
Out[50]: array([ 1.47586906,  1.48365788])
In [51]: f(x_opt)
Out[51]: -9.52022927306
```

Here, the BFGS method gave the final minimum point $(x_1, x_2) = (1.47586906, 1.48365788)$, with the minimum value of the objective function –9.52022927306. For this type of problem, guessing the initial starting point easily results in the iterative solver converging to a local minimum, and the systematic approach that `optimize.brute` provides is frequently useful.

As always, it is important to visualize the objective function and the solution when possible. The following code listings plot a contour graph of the current objective function and mark the obtained solution with a red star (see Figure 6-5). As in the previous example, we need a wrapper function for reshuffling the parameters of the objective function because of the different conventions of how the coordinated vectors are passed to the function (separate arrays and packed into one array, respectively).

```
In [52]: def func_X_Y_to_XY(f, X, Y):
    ...:     """
    ...:     Wrapper for f(X, Y) -> f([X, Y])
    ...:     """
    ...:     s = np.shape(X)
    ...:     return f(np.vstack([X.ravel(), Y.ravel()])).reshape(*s)
In [53]: fig, ax = plt.subplots(figsize=(6, 4))
    ...: x_ = y_ = np.linspace(-3, 5, 100)
    ...: X, Y = np.meshgrid(x_, y_)
    ...: c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 25)
    ...: ax.plot(x_opt[0], x_opt[1], 'r*', markersize=15)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: plt.colorbar(c, ax=ax)
```

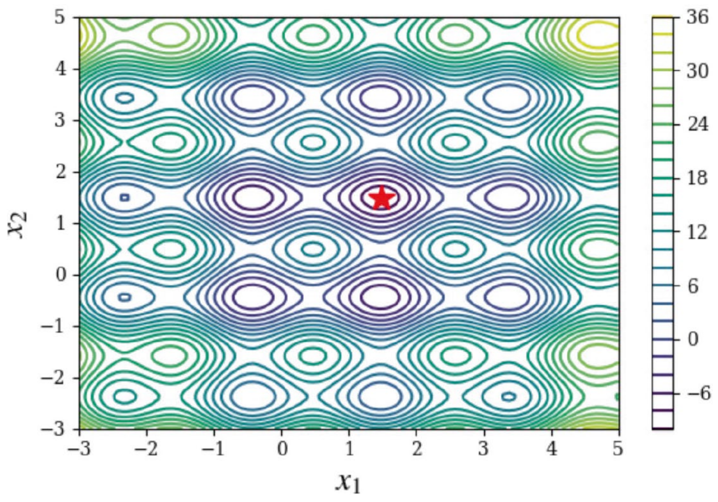

Figure 6-5. *Contour plot of the objective function f(x) = 4 sin xπ + 6 sin yπ + (x − 1)^2 + (y − 1)^2. The minimum is marked with a red star*

This section explicitly calls functions for specific solvers, for example, `optimize.fmin_bfgs`. However, like for scalar optimization, SciPy also provides a unified interface for all multivariate optimization solvers with the `optimize.minimize` function, which dispatches out to the solver-specific functions depending on the value of the `method` keyword argument (remember, the univariate minimization function that provides a unified interface is `optimize.scalar_minimize`). For clarity, here I have favored explicitly calling functions for specific solvers, but in general, it is a good idea to use `optimize.minimize`, as this makes it easier to switch between different solvers. For example, the previous example used `optimize.fmin_bfgs` in the following way.

```
In [54]: x_opt = optimize.fmin_bfgs(f, x_start)
```

We could just as well have used the following.

```
In [55]: result = optimize.minimize(f, x_start, method='BFGS')
In [56]: x_opt = result.x
```

The `optimize.minimize` function returns an instance of `optimize.OptimizeResult` that represents the result of the optimization. The solution is available via the `x` attribute of this class.

Nonlinear Least Square Problems

In Chapter 5, we encountered linear least square problems and explored how they can be solved using linear algebra. In general, a least square problem can be viewed as an optimization problem with the objective function $g(\beta)=\sum_{i=0}^{m} r_i(\beta)^2 = \|r(\beta)\|^2$, where $r(\beta)$ is a vector with the residuals $r_i(\beta) = y_i - f(x_i, \beta)$ for a set of m observations (x_i, y_i). Here, β is a vector with unknown parameters that specifies the $f(x, \beta)$ function. If this problem is nonlinear in the parameters β, it is known as a nonlinear least square problem, and since it is nonlinear, it cannot be solved with the linear algebra techniques discussed in Chapter 5. Instead, we can use the multivariate optimization techniques described in the previous section, such as Newton's or quasi-Newton methods. However, this nonlinear least square optimization problem has a specific structure, and several methods have been developed to solve this particular optimization problem. One example is the Levenberg-Marquardt method, which is based on successive linearizations of the problem in each iteration.

In SciPy, the `optimize.leastsq` function provides a nonlinear least square solver that uses the Levenberg-Marquardt method. To illustrate how this function can be used, consider a nonlinear model on the form $f(x, \beta) = \beta_0 + \beta_1 \exp(-\beta_2 x^2)$ and a set of observations (x_i, y_i). The following example simulates the observations with random noise added to the actual values and solves the minimization problem that gives the best least square estimates of the parameters β. First, we define a tuple with the actual values of the parameter vector β and a Python function for the model function. This function, which should return the y value corresponding to a given x value, takes as the first argument the variable x, and the following arguments are the unknown function parameters.

```
In [57]: beta = (0.25, 0.75, 0.5)
In [58]: def f(x, b0, b1, b2):
    ...:    return b0 + b1 * np.exp(-b2 * x**2)
```

Once the model function is defined, generate randomized data points that simulate the observations.

```
In [59]: xdata = np.linspace(0, 5, 50)
In [60]: y = f(xdata, *beta)
In [61]: ydata = y + 0.05 * np.random.randn(len(xdata))
```

We can start solving the nonlinear least square problem with the model function and observation data prepared. The first step is to define a function for the residuals given the data and the model function specified in terms of the yet-to-be-determined model parameters β.

```
In [62]: def g(beta):
    ...:     return ydata - f(xdata, *beta)
```

Next, define an initial guess for the parameter vector and let the `optimize.leastsq` function solves for the best least square fit for the parameter vector.

```
In [63]: beta_start = (1, 1, 1)
In [64]: beta_opt, beta_cov = optimize.leastsq(g, beta_start)
In [65]: beta_opt
Out[65]: array([ 0.25733353,  0.76867338,  0.54478761])
```

The best fit is close to the actual parameter values (0.25, 0.75, 0.5), as defined earlier. By plotting the observation data and the model function for the actual and fitted function parameters, we can visually confirm that the fitted model describes the data well (see Figure 6-6).

```
In [66]: fig, ax = plt.subplots()
    ...: ax.scatter(xdata, ydata, label='samples')
    ...: ax.plot(xdata, y, 'r', lw=2, label='true model')
    ...: ax.plot(xdata, f(xdata, *beta_opt), 'b', lw=2, label='fitted model')
    ...: ax.set_xlim(0, 5)
    ...: ax.set_xlabel(r"$x$", fontsize=18)
    ...: ax.set_ylabel(r"$f(x, \beta)$", fontsize=18)
    ...: ax.legend()
```

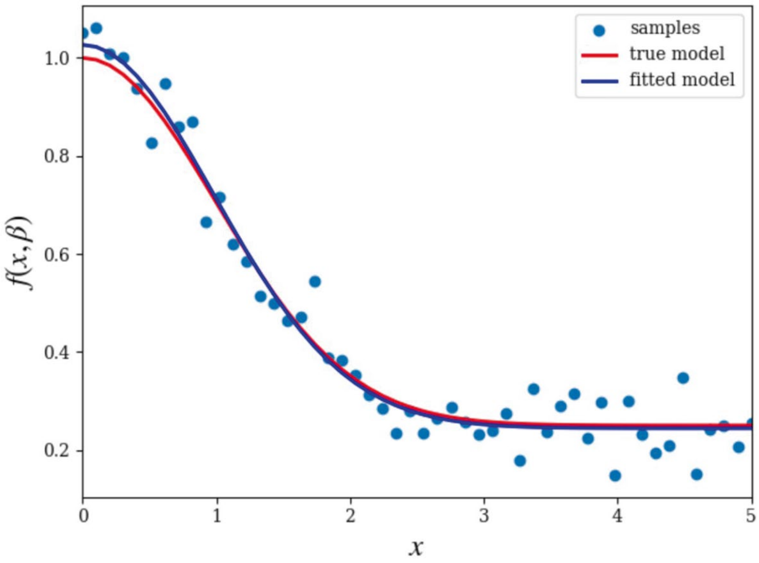

***Figure 6-6.** Nonlinear least square fit to the $f(x, \beta) = \beta_0 + \beta_1 \exp(-\beta_2 x^2)$ function with $\beta = (0.25, 0.75, 0.5)$*

The SciPy `optimize` module provides an alternative interface to nonlinear least square fitting through the `optimize.curve_fit` function. This convenience wrapper around `optimize.leastsq` eliminates the need to explicitly define the residual function for the least square problem. The previous problem could, therefore, be solved more concisely using the following.

```
In [67]: beta_opt, beta_cov = optimize.curve_fit(f, xdata, ydata)
In [68]: beta_opt
Out[68]: array([ 0.25733353,  0.76867338,  0.54478761])
```

Constrained Optimization

Constraints add another level of complexity to optimization problems, requiring their own classification. A simple form of constrained optimization is where the coordinate variables are subject to some bounds; for example, $\min_x f(x)$ subject to $0 \leq x \leq 1$. The constraint $0 \leq x \leq 1$ is simple because it only restricts the range of the coordinate without dependencies on the other variables. This type of problem can be solved using the L-BFGS-B method in SciPy, a variant of the BFGS method used earlier. This solver is available through the `optimize.fmin_l_bgfs_b` function or via `optimize.minimize` with the `method` argument set to `'L-BFGS-B'`. To define the coordinate boundaries, the `bound` keyword argument must be used, and its value should be a list of tuples that contain the minimum and maximum value of each constrained variable. If the minimum or maximum value is set to `None`, it is interpreted as an unbounded.

As an example of solving a bounded optimization problem with the L-BFGS-B solver, consider minimizing the objective function $f(x) = (x_1 - 1)^2 - (x_2 - 1)^2$ subject to the constraints $2 \leq x_1 \leq 3$ and $0 \leq x_2 \leq 2$. First, to solve this problem, define a Python function for the objective functions and tuples with the boundaries for each of the two variables in this problem, according to the given constraints. For comparison, the following code also solves the unconstrained optimization problem with the same objective function and plots a contour graph of the objective function where the unconstrained and constrained minimum values are marked with blue and red stars, respectively (see Figure 6-7).

```
In [69]: def f(X):
    ...:     x, y = X
    ...:     return (x - 1)**2 + (y - 1)**2
In [70]: x_opt = optimize.minimize(f, [0, 0], method='BFGS').x
In [71]: bnd_x1, bnd_x2 = (2, 3), (0, 2)
In [72]: x_cons_opt = optimize.minimize(f, [0, 0], method='L-BFGS-B',
    ...:                                bounds=[bnd_x1, bnd_x2]).x
In [73]: fig, ax = plt.subplots(figsize=(6, 4))
    ...: x_ = y_ = np.linspace(-1, 3, 100)
    ...: X, Y = np.meshgrid(x_, y_)
    ...: c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 50)
    ...: ax.plot(x_opt[0], x_opt[1], 'b*', markersize=15)
    ...: ax.plot(x_cons_opt[0], x_cons_opt[1], 'r*', markersize=15)
    ...: bound_rect = plt.Rectangle(
    ...:     (bnd_x1[0], bnd_x2[0]),
    ...:     bnd_x1[1] - bnd_x1[0], bnd_x2[1] - bnd_x2[0], facecolor="grey")
    ...: ax.add_patch(bound_rect)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: plt.colorbar(c, ax=ax)
```

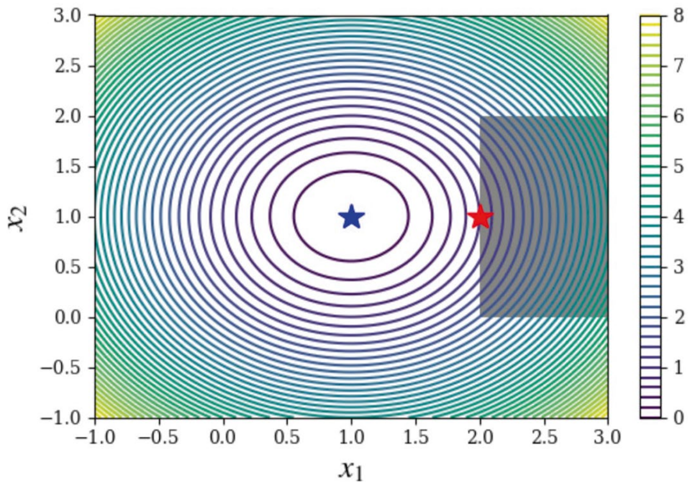

Figure 6-7. *Contours of the objective function f(x), with the unconstrained (blue star) and constrained minima (red star). The feasible region of the constrained problem is shaded in gray*

Constraints defined by equalities or inequalities that include more than one variable are somewhat more complicated. However, there are general techniques also for this type of problem. For example, it is possible to convert a constrained optimization problem to an unconstrained problem using the Lagrange multipliers by introducing additional variables. For example, consider the optimization problem $\min_x f(x)$ methods, linear programming problems with thousands $g(x) = 0$. In an unconstrained optimization problem, the gradient of $f(x)$ vanishes at the optimal points, $\nabla f(x) = 0$. It can be shown that the corresponding condition for constrained problems is that the negative gradient lies in the space supported by the constraint normal, such as $-\nabla f(x) = \lambda J_g^T(x)$. Here, $J_g(x)$ is the Jacobian matrix of the constraint function $g(x)$, and λ is the vector of Lagrange multipliers (new variables). This condition arises from equating to zero the gradient of the $\Lambda(x, \lambda) = f(x) + \lambda^T g(x)$ function, which is known as the Lagrangian function. Therefore, if both $f(x)$ and $g(x)$ are continuous and smooth, a stationary point (x_0, λ_0) of the $\Lambda(x, \lambda)$ function corresponds to an x_0 that is an optimum of the original constrained optimization problem. Note that if $g(x)$ is a scalar function (i.e., there is only one constraint), then the Jacobian $J_g(x)$ reduces to the gradient $\nabla g(x)$.

To illustrate this technique, consider the problem of maximizing the volume of a rectangle with sides of length x_1, x_2, and x_3, subject to the constraint that the total surface area should be unity: $g(x) = 2x_1x_2 + 2x_0x_2 + 2x_1x_0 - 1 = 0$. To solve this optimization problem using Lagrange multipliers, form the Lagrangian $\Lambda(x) = f(x)+\lambda g(x)$ and seek the stationary points for $\nabla\Lambda(x) = 0$. With SymPy, we can carry out this task by first defining the symbols for the variables in the problem, then constructing expressions for $f(x)$, $g(x)$, and $\Lambda(x)$.

```
In [74]: x = x0, x1, x2, l = sympy.symbols("x_0, x_1, x_2, lambda")
In [75]: f = x0 * x1 * x2
In [76]: g = 2 * (x0 * x1 + x1 * x2 + x2 * x0) - 1
In [77]: L = f + l * g
```

Finally, compute $\nabla\Lambda(x)$ using `sympy.diff` and solve the equation $\nabla\Lambda(x) = 0$ using `sympy.solve`.

```
In [78]: grad_L = [sympy.diff(L, x_) for x_ in x]
In [79]: sols = sympy.solve(grad_L)
In [80]: sols
```

Out[80]: $$\left[\left\{\lambda : -\frac{\sqrt{6}}{24}, x_0 : \frac{\sqrt{6}}{6}, x_1 : \frac{\sqrt{6}}{6}, x_2 : \frac{\sqrt{6}}{6}\right\}, \left\{\lambda : \frac{\sqrt{6}}{24}, x_0 : -\frac{\sqrt{6}}{6}, x_1 : -\frac{\sqrt{6}}{6}, x_2 : -\frac{\sqrt{6}}{6}\right\}\right]$$

This procedure gives two stationary points. We could determine which one corresponds to the optimal solution by evaluating the objective function for each case. However, here, only one of the stationary points corresponds to a physically acceptable solution: since x_i is the length of a rectangle side in this problem, it must be positive. We can, therefore, immediately identify the interesting solution corresponding to the intuitive result $x_0 = x_1 = x_2 = \sqrt{6}/6$ (a cube). As a final verification, evaluate the constraint function and the objective function using the obtained solution.

```
In [81]: g.subs(sols[0])
Out[81]: 0
In [82]: f.subs(sols[0])
```

Out[82]: $\frac{\sqrt{6}}{36}$

The method of using Lagrange multipliers can also be extended to handle inequality constraints, and several numerical approaches apply such techniques. One example is the sequential least square programming method, abbreviated as SLSQP, which is available in the SciPy as the `optimize.slsqp` function and via `optimize.minimize` with `method='SLSQP'`. The `optimize.minimize` function takes the keyword argument `constraints`, which should be a list of dictionaries that each specifies a constraint. The allowed keys (values) in this dictionary are `type` (`'eq'` or `'ineq'`), `fun` (constraint function), `jac` (Jacobian of the constraint function), and `args` (additional arguments to constraint function and the function for evaluating its Jacobian). For example, the constraint dictionary describing the constraint in the previous problem would be `dict(type='eq', fun=g)`.

To solve the full problem numerically using SciPy's SLSQP solver, we need to define Python functions for the objective function and the constraint function.

```
In [83]: def f(X):
    ...:     return -X[0] * X[1] * X[2]
```

```
In [84]: def g(X):
    ...:     return 2 * (X[0]*X[1] + X[1] * X[2] + X[2] * X[0]) - 1
```

Note that since the SciPy optimization functions solve minimization problems, and here we are interested in maximization, the `f` function is the negative of the original objective function. Next, let's define the constraint dictionary for $g(x) = 0$ and finally call the `optimize.minimize` function.

```
In [85]: constraint = dict(type='eq', fun=g)
In [86]: result = optimize.minimize(f, [0.5, 1, 1.5], method='SLSQP',
    ...:                            constraints=[constraint])
In [87]: result
Out[87]:  status: 0
         success: True
            njev: 18
            nfev: 95
             fun: -0.068041368623352985
               x: array([ 0.40824187,  0.40825127,  0.40825165])
         message: 'Optimization terminated successfully.'
             jac: array([-0.16666925, -0.16666542, -0.16666527,  0.])
             nit: 18
In [88]: result.x
Out[88]: array([ 0.40824187,  0.40825127,  0.40825165])
```

As expected, the solution agrees well with the analytical result of the symbolic calculation using Lagrange multipliers.

To solve problems with inequality constraints, we must set `type='ineq'` in the constraint dictionary and provide the corresponding inequality function. To demonstrate the minimization of a nonlinear objective function with a nonlinear inequality constraint, return to the quadratic problem considered previously, but in this case, with inequality constraint $g(x) = x_1 - 1.75 - (x_0 - 0.75)^4 \geq 0$. As usual, begin by defining the objective function, the constraint function, and the constraint dictionary.

```
In [89]: def f(X):
    ...:     return (X[0] - 1)**2 + (X[1] - 1)**2
In [90]: def g(X):
    ...:     return X[1] - 1.75 - (X[0] - 0.75)**4
In [91]: constraints = [dict(type='ineq', fun=g)]
```

Next, we can solve the optimization problem by calling the `optimize.minimize` function. For comparison, let's also solve the corresponding unconstrained problem.

```
In [92]: x_opt = optimize.minimize(f, (0, 0), method='BFGS').x
In [93]: x_cons_opt = optimize.minimize(f, (0, 0), method='SLSQP',
constraints=constraints).x
```

To verify the soundness of the obtained solution, we plot the contours of the objective function together with a shaded area representing the feasible region (where the inequality constraint is satisfied). The constrained and unconstrained solutions are marked with a red and a blue star, respectively (see Figure 6-8).

```
In [94]: fig, ax = plt.subplots(figsize=(6, 4))
In [95]: x_ = y_ = np.linspace(-1, 3, 100)
    ...: X, Y = np.meshgrid(x_, y_)
```

```
   ...: c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 50)
   ...: ax.plot(x_opt[0], x_opt[1], 'b*', markersize=15)
   ...: ax.plot(x_, 1.75 + (x_-0.75)**4, 'k-', markersize=15)
   ...: ax.fill_between(x_, 1.75 + (x_-0.75)**4, 3, color='grey')
   ...: ax.plot(x_cons_opt[0], x_cons_opt[1], 'r*', markersize=15)
   ...:
   ...: ax.set_ylim(-1, 3)
   ...: ax.set_xlabel(r"$x_0$", fontsize=18)
   ...: ax.set_ylabel(r"$x_1$", fontsize=18)
   ...: plt.colorbar(c, ax=ax)
```

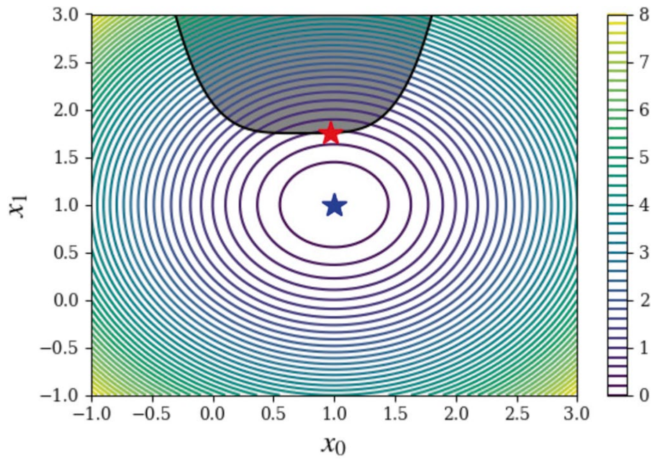

Figure 6-8. *Contour plot of the objective function with the feasible region of the constrained problem shaded gray. The red and blue stars are the optimal points in the constrained and unconstrained problems, respectively*

For optimization problems with *only* inequality constraints, SciPy provides an alternative solver using the constrained optimization by linear approximation (COBYLA) method. This solver is accessible through `optimize.fmin_cobyla` or `optimize.minimize` with `method='COBYLA'`. It could have solved the previous example by replacing `method='SLSQP'` with `method='COBYLA'`.

Linear Programming

The previous section considered methods for general optimization problems where the objective and constraint functions can be nonlinear. However, at this point, it is worth taking a step back and considering a much more restricted type of optimization problem, namely, *linear programming*, where the objective function is linear, and all constraints are linear equality or inequality constraints. The class of problems is much less general. But it turns out that linear programming has many important applications that can be solved vastly more efficiently than general nonlinear problems. This is because linear problems have properties that enable entirely different methods. In particular, the solution to a linear optimization problem must necessarily lie on a constraint boundary, so it is sufficient to search the vertices of the intersections of the linear constraint functions. This can be done efficiently in practice. A popular algorithm for this type of problem is known as *simplex*, which systematically moves from one vertex to another until the optimal vertex has been reached. More recent interior point methods also efficiently solve linear programming problems. These methods make linear programming problems with thousands of variables and constraints readily solvable.

Linear programming problems are typically written in the standard form: min$xcTx$ where $Ax \leq b$ and $x \geq 0$. Here, c and x are vectors of length n, A is a $m \times n$ matrix, and b is a m-vector. For example, consider the problem of minimizing the $f(x) = -x_0 + 2x_1 - 3x_2$ function, subject to the three inequality constraints $x_0 + x_1 \leq 1$, $-x_0 + 3x_1 \leq 2$, and $-x_1 + x_2 \leq 3$. The standard form has $c = (-1, 2, -3)$, $b = (1, 2, 3)$, and

$$A = \begin{pmatrix} 1 & 1 & 0 \\ -1 & 3 & 0 \\ 0 & -1 & 1 \end{pmatrix}.$$

To solve this problem, use the `cvxopt` library, which provides the linear programming solver with the `cvxopt.solvers.lp` function. This solver expects the c, A, and b vectors and matrix used in the standard form introduced in the preceding text in the given order as arguments. The `cvxopt` library uses its own classes for representing matrices and vectors. Fortunately, they are interoperable with NumPy arrays via the array interface[3] and can, therefore, be cast from one form to another using the `cvxopt.matrix` and `np.array` functions. Since the NumPy array is the de facto standard array format in the scientific Python environment, it is sensible to use the NumPy array as far as possible and only convert to cvxopt matrices when necessary (i.e., before calling one of the solvers in `cvxopt.solvers`).

To solve the stated example problem using the `cvxopt` library, first, create NumPy arrays for the A matrix and the c and b vectors and convert them to `cvxopt` matrices using the `cvxopt.matrix` function.

```
In [96]: c = np.array([-1.0, 2.0, -3.0])
In [97]: A = np.array([[ 1.0, 1.0, 0.0],
                       [-1.0, 3.0, 0.0],
                       [ 0.0, -1.0, 1.0]])
In [98]: b = np.array([1.0, 2.0, 3.0])
In [99]: A_ = cvxopt.matrix(A)
In [100]: b_ = cvxopt.matrix(b)
In [101]: c_ = cvxopt.matrix(c)
```

The `cvxopt` compatible matrices and vectors `c_`, `A_`, and `b_` can now be passed to the linear programming solver `cvxopt.solvers.lp`.

```
In [102]: sol = cvxopt.solvers.lp(c_, A_, b_)
In [103]: sol
Out[103]: {'dual infeasibility': 1.4835979218054372e-16,
           'dual objective': -10.0,
           'dual slack': 0.0,
           'gap': 0.0,
           'iterations': 0,
           'primal infeasibility': 0.0,
           'primal objective': -10.0,
           'primal slack': -0.0,
           'relative gap': 0.0,
           'residual as dual infeasibility certificate': None,
           'residual as primal infeasibility certificate': None,
           's': <3x1 matrix, tc='d'>,
           'status': 'optimal',
           'x': <3x1 matrix, tc='d'>,
```

[3] For details, see http://docs.scipy.org/doc/numpy/reference/arrays.interface.html.

```
           'y': <0x1 matrix, tc='d'>,
           'z': <3x1 matrix, tc='d'>}
In [104]: x = np.array(sol['x'])
In [105]: x
Out[105]: array([[ 0.25],
                 [ 0.75],
                 [ 3.75]])
In [106]: sol['primal objective']
Out[106]: -10.0
```

The solution to the optimization problem is given in terms of the vector x, which in this example is $x = (0.25, 0.75, 3.75)$, which corresponds to the $f(x)$ value -10. With this method and the `cvxopt.solvers.lp` solver, linear programming problems with hundreds or thousands of variables can readily be solved. All that is needed is to write the optimization problem on the standard form and create the c, A, and b arrays.

Summary

Optimization—selecting the best option from a set of alternatives—is fundamental in many applications in science and engineering. Mathematical optimization provides a rigorous framework for systematically treating optimization problems if they can be formulated as mathematical problems. Computational optimization methods are the tools for solving such optimization problems in practice. In a scientific computing environment, optimization, therefore, plays a very important role. For scientific computing with Python, the SciPy library provides efficient routines for solving many standard optimization problems, which can be used to solve many computational optimization problems. However, optimization is a large field in mathematics, requiring different methods for solving different types of problems, and several optimization libraries for Python provide specialized solvers to specific kinds of optimization problems. The SciPy `optimize` module offers excellent and flexible general-purpose solvers for a wide variety of optimization problems. But, for particular optimization problems, many specialized libraries provide better performance or more features. An example of such a library is `cvxopt`, which complements the general-purpose optimization routines in SciPy with efficient solvers for linear and quadratic problems.

Further Reading

For an accessible introduction to optimization, with more detailed discussions of the numerical properties of several of the methods introduced in this chapter, see, for example, *Scientific Computing: An Introductory Survey* by M. Heath (McGraw-Hill, 2002). For a more rigorous and in-depth introduction to optimization, see, for example, *An Introduction to Optimization* by E. K. P. Chong (Wiley, 2013). The creators of the `cvxopt` library give a thorough treatment of convex optimization in the excellent book *Convex Optimization* by S. Boyd (Cambridge University Press, 2004), which is also available online at http://stanford.edu/~boyd/cvxbook.